3.1 Overview

To enable diffusion-based generation of ASTs from natural language instructions, we propose a data transformation pipeline that constructs a paired text-to-AST dataset from raw source code. The overall workflow is illustrated in Figure 1.

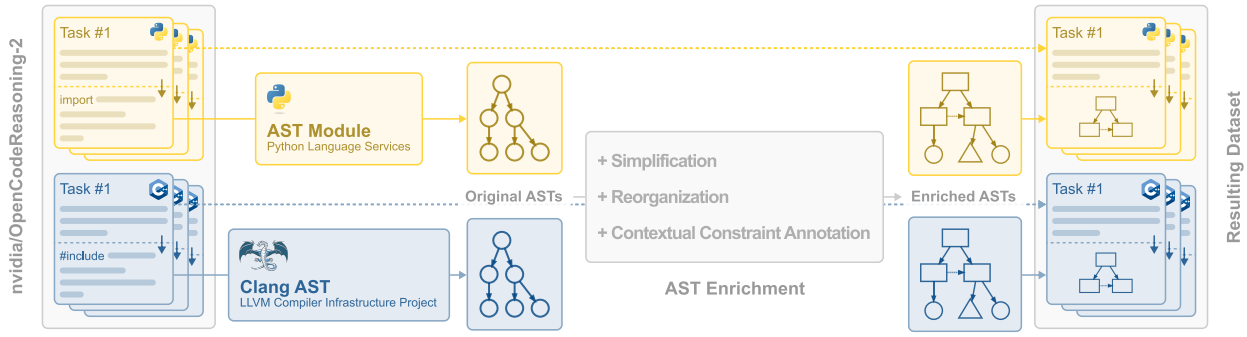


Figure 1: Pipeline overview

We begin with the `nvidia/OpenCodeReasoning-2` dataset [24], which provides a large number of programming task descriptions and corresponding code snippets in Python and C++. For each language, we parse the source code using off-the-shelf language infrastructure—Python Language Services and Clang AST—to obtain the original ASTs. These ASTs are then passed through an enrichment module, which performs:

- **Simplification** of redundant structural element and **reduction** of unnecessary AST depth,
- **Reorganization** of node layout for consistent and canonical structure,
- **Contextual constraint annotation** to embed program-level contextual semantics.

The result is a dataset that pairs each textual task description with an enriched AST, suitable for training diffusion models that capture both syntactic and semantic regularities of real-world programs.

3.2 Text-to-code Dataset Selection

For the design and implementation of this prototype, we selected the `nvidia/OpenCodeReasoning-2` dataset. This dataset offers large-scale, task-oriented examples in two widely-used programming:

- **Python**: ∼1.40 million code solutions for 34,125 tasks
- **C++**: ∼1.17 million code solutions for 30,092 tasks
- **Total**: ∼2.57 million instances covering diverse competitive programming tasks

Each task-solution pair is stored in the dataset under the following format:

Table 1: Data format of dataset `nvidia/OpenCodeReasoning-2`

Column	Description
id	A unique id for each data instance
question_id	A unique id for each question
question	The input competitive programming question.
solution	The code portion of R1's response.
dataset	The name of the dataset from which this question is collected from.
split	The name of the split of the dataset from which this question is collected from.
index	An index to retrieve the input question from APPS/TACO dataset.

Table 1 summarizes the relevant fields used in our pipeline. The `question` field provides the natural language problem description. Due to copyright limitations, the field is left blank in the dataset. However, the authors provide a Python script to look up the questions from the task datasets using the `dataset` field and

`index` field. The `solution` field contains the corresponding code implementation. Additional metadata such as `question_id` and `split` helps organize and trace the provenance of each example across source datasets like APPS and TACO.

The above dataset is particularly well-suited for this project as a starting point due to its scale and task-oriented structure. With over two million examples across Python and C++, it offers a diverse collection of real-world programming problems paired with corresponding code solutions. This makes it an ideal foundation for learning mappings from natural language instructions to structured code representations.

The contrast between a strongly typed language like C++ and a dynamically typed language like Python provides two distinct design contexts for our prototype. C++ demands precise type resolution and enforces rigid structural rules, while Python allows greater flexibility with implicit typing and similar syntax. By supporting both, our pipeline must accommodate fundamentally different AST characteristics and enrichment requirements. This dual-language prototype setup not only validates the generality of our approach but also offers practical insights for extending the system to additional languages in the future.

3.3 Structural Representations of Programming Languages

To extract structured representations from the source code, we employ existing language-specific AST parsers for Python and C++. These parsers allow us to convert raw code into ASTs, which serve as the basis for further enrichment and dataset construction.

For Python, we adopt the built-in `ast` module from the standard library, which provides a concrete AST representation of parsed Python source code. To convert the AST into a machine readable format, we use the third-party `ast2json` utility library, which serializes the AST into a nested JSON format. Each node is represented as a JSON object whose type is marked by its field "`_type`", along with fields corresponding to its children and other attributes. Figure 2 demonstrates the major hierarchical structure of certain types of node in an AST, including assignment statement, `for`-loop statement, and function call expression.

```
{
  "_type": "Assign",
  "targets": [ /* targets */ ],
  "value": { /* value */ },
  // Others
}
```

(a) Assignment statement

```
{
  "_type": "For",
  "target": { /* target */ },
  "iter": { /* iterator */ },
  "body": [ /* elements */ ],
  // Others
}
```

(b) For statement

```
{
  "_type": "Call",
  "func": { /* function */ },
  "args": [ /* arguments */ ],
  // Others
}
```

(c) Function call expression

Figure 2: Inner structures of certain types of node in a Python AST

This JSON-based representation of Python ASTs is relatively clean and consistent, making it easy to traverse, manipulate and enrich. The explicit labeling of node types and clear hierarchical organization facilitate downstream processing tasks such as normalization, constraint extraction and serialization. Moreover, the Python AST is relatively compact and semantically transparent, owing to the language's syntactic simplicity and dynamic nature.

In contrast, C++ presents a much more complex landscape. Its statistically typed nature, intricate grammar and reliance on compile-time constructs, *e.g.*, templates, operator overloading, type deduction, result in ASTs that are significantly deeper, more verbose and less uniform. As a result, while Clang's JSON output provides comprehensive syntactic and semantic information, its structure tends to be harder to interpret and more difficult to normalize directly. This complexity necessitates additional pre-processing and post-processing steps to extract meaningful and consistent representations across code samples.

As a part of the LLVM Compiler Infrastructure Project, Clang front-end supports exporting structured AST in JSON format via the `-Xclang -ast-dump=json` option. Clang performs full semantic analysis during parsing, enabling the extraction of detailed type information, declaration scopes and template instantiations. This makes it an ideal tool for constructing rich, context-aware ASTs, while also leaving us a few challenges.

A notable compatibility issue in pre-processing C++ code is the frequent use of the non-standard header `bits/stdc++.h`, which is common in competitive programming due to its convenience in aggregating most standard library headers. However, this header is GCC-specific and not supported by Clang, resulting in fatal errors during parsing. To resolve this, we implement a normalization step that replaces `#include<bits/stdc++.h>` with an explicit list of standard headers before invoking Clang. This transformation preserves program semantics while significantly improving the pipeline's ability to process a broader range of C++ examples in the dataset.

Another major challenge arises from the structural complexity of Clang's AST output. Although, like Python, each node in the JSON is labeled by its "`kind`" field indicating its syntactic category, Clang often collapses the internal structure of a syntax node into a generic "`inner`" list without explicitly distinguishing the roles of its children. As a result, it is often impossible to infer the semantic function of sub-nodes from the JSON alone, *e.g.*, whether a child represents the initializer, the condition, the increment, or the body part of a `for`-loop statement.

```
{
  "kind": "IfStmt",
  "inner": [
    { /* condition */ }, // 1
    { /* condition variable declaration */ }, // 2
    // Either 1 or 2 can be present
    { /* then statement */ },
    { /* else statement */ } // May or may not exist
  ],
  // Others
}
```

(a) If statement

```
{
  "kind": "ForStmt",
  "inner": [
    { /* initializer */ }, // 1
    { /* condition variable declaration */ }, // 2
    // Either 1 or 2 is empty
    { /* condition expression */ },
    { /* increment */ },
    { /* body statement */ }
  ],
  // Others
}
```

(b) For statement

Figure 3: Inner structures of certain types of node in a C++ AST

To address this, we conducted an in-depth study of Clang's official documentation and source code, combined with numeral examples of the ASTs extracted with Clang, to understand the internal AST design. Figure 3 reveals the major hierarchical structures of `if`-statement and `for`-statement in the C++ ASTs. This effort informed the development of our enriched AST schema, which aims to explicitly capture as many as C++ language constructs and roles as possible during the transformation and enrichment process.

To summarize, these structural differences between Python and C++ AST representations highlight the need for language-specific handling in our pipeline. By leveraging existing parsers and applying targeted normalization and identification of internal structures, further semantic enrichment can be applied to our unified structural foundation for either language.

3.4 Enriched AST Schema Design

While raw ASTs capture the syntactic structure of code, they often lack sufficient semantic context for high-quality program understanding and generation. To bridge the gap, we design and implement an enriched AST schema that augments the baseline trees with additional program-level information—such as type annotations, variable lifecycles and available APIs—collected through static analysis. This enriched representation provides the model with deeper insight into program semantics and improves its ability to generate well-formed code structures.

Given the significant differences between C++ and Python in type systems, scoping rules, compilation models and AST structures, we design language-specific enrichment procedures tailored to each language. The remainder of this part outlines our approach for the languages individually.

3.4.1 Enriched AST Schema for C++

Clang exposes a highly detailed and expansive AST type system for C++, covering the full breadth of the language's syntax and semantics. Given this complexity, implementing a comprehensive resolution pipeline that handles every possible AST node is impractical. To address this, we adopt a data-driven incremental strategy grounded in empirical observation of real-world code.

We begin by sampling a small subset of C++ programs from the dataset and manually analyzing the Clang ASTs generated from them. Based on this sample, we identify the syntactic structures present in the ASTs, which can be categorized into statements, expressions and declarations, and implement corresponding handlers in our enrichment pipeline. We then expand the sampling range and process new programs using the existing implementation. For any parseable code samples that contain yet-unhandled node types, we collect them and progressively extend our support to cover these new constructs. This iterative process continues until the system achieves a stable coverage across a significant portion of the dataset.

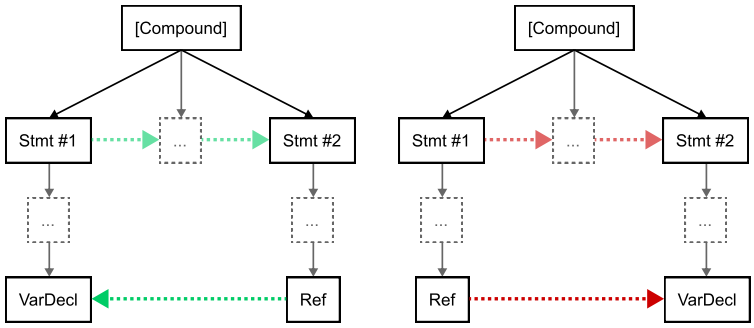
To keep the enrichment process focused and maintainable, we constrain our analysis to the body of the `main` function in each program. This choice is motivated primarily by implementation complexity: supporting the

full range of C++ constructs—including classes, templates and lambda expressions—would require substantial engineering effort beyond the scope of this project. In contrast, the `main` function typically contains sufficient structural variety to exercise key aspects of our enrichment scheme, including control flow, function calls, variable declarations and references, and type resolution. By limiting out target to this scope, we reduce parsing and normalization complexity while retaining a meaningful subset of C++ program semantics.

One of the central components of our AST enrichment strategy is type annotation. Fortunately, Clang performs comprehensive type inference and resolution during parsing, and encodes this information directly into the JSON-formatted AST. For most syntax nodes, Clang provides a qualified type, *e.g.*, with `const` or reference modifiers, and/or a desugared type, which resolves `typedef`'s and syntactic sugar to their canonical forms. This built-in support significantly reduces the burden of implementing custom type analysis. In our enrichment pipeline, these annotations are included as explicit fields in the enriched schema, where desugared types are prioritized over qualified types to maintain canonical consistency across nodes. This type information serves as a crucial inductive bias during model training, helping ensure that generated ASTs are not only syntactically valid but also type-consistent.

Another key aspect of our enrichment schema is the incorporation of statement execution order. While ASTs represent code as hierarchical tree structures, compound statements, *e.g.*, function bodies or code blocks, are typically modeled as unordered lists of child nodes. However, in imperative programming languages like C++, the order of statements is semantically critical: it determines data dependencies, variable life cycles and control flow semantics. To capture this sequential aspect explicitly, we introduce virtual edges in the enriched AST to represent the execution order among statements within compound statement structures. These edges connect sibling nodes to reflect their original order in the source code, without altering the underlying tree structure. By augmenting the ASTs in this way, we enable downstream models to incorporate sequential reasoning over statement execution, which is essential for tasks such as variable reference resolution and instruction reordering.

In order to capture data flow and identifier resolution within a program, we enrich the AST with explicit variable reference links. When compiling a C++ source file, Clang creates a unique `VarDecl` node for each variable declaration. All subsequent references to that variable—whether in expressions, assignments, or control conditions—are internally linked to the same memory-resident declaration node during the compilation process. In the JSON-formatted ASTs exported by Clang, this internal linkage is preserved via a unique `id` field assigned to each node. Reference nodes, *e.g.*, `DeclRefExpr`, contain a `referencedDecl` field pointing to the `id` of the corresponding `VarDecl`. We leverage this mechanism to add virtual reference edges that explicitly connect each reference to its declared variable. These edges are included in the enriched AST structure to expose variable bindings and usage scopes, enabling downstream models to reason over read/write dependencies and identifier resolution more effectively.



(a) Valid reference (b) Invalid reference

Figure 4: An example of reference validity check

By combining the sequential virtual edges with variable reference links, the enriched ASTs enable static validation of reference legality. Specifically, when validate a variable reference, we trace both the reference node and its corresponding declaration node upward the in the AST until they converge at a common enclosing compound statement. Within this compound structure, we then compare their positions in the sequential execution order of child statements. A reference is considered semantically valid only if the declaration node appears earlier than the reference node within the compound statement's internal execution sequence. This check ensures that all variable usages respect lexical and control-flow ordering constraints. Figure 4 shows an example where we apply such strategy to checking the reference validity.

Finally, one of the most challenging aspects of processing C++ ASTs lies in handling references to non-user-

defined entities, particularly those introduced via standard library headers. Unlike user-defined functions and variables, which typically exhibit simple and localized structure, standard headers such as `<iostream>` bring in a vast amount of additional code. When compiled with Clang, these headers are fully expanded and embedded into the AST, resulting in extreme verbosity. For example, including `<iostream>` alone can lead to a JSON AST exceeding 400MB in size and spanning nearly 8 million lines—most of which are irrelevant to the user's original program logic.

To manage this complexity, our current pipeline focuses exclusively on user-defined declarations. All references to standard library functions or variables are marked as external references, without attempting to resolve their full definitions or include them in the enriched AST. This design choice significantly reduces memory usage and processing time, while retaining the essential program structure necessary for our modeling tasks.

Overall, our C++ AST enrichment module combines syntax-guided filtering, type annotation extraction, execution-order reconstruction, and variable reference resolution to produce a semantically aware and structurally consistent representation. By carefully balancing coverage and complexity, we construct an enriched AST schema that preserves essential program semantics while significantly reducing the size of the representations,making it practical for large-scale generative modeling.

3.4.2 Enriched AST Schema for Python

Compared to C++, Python is a dynamically typed scripting language with minimal compile-time type enforcement. As a result, its native AST representation lacks many of the structural and semantic constraints that are crucial for static code analysis and program understanding. In particular, the standard `ast` module provides only a shallow syntactic view of the code and does not include type information or resolved symbol references.

To overcome these limitations, we adopt the `astroid` library [25] from the `pylint` toolchain as a replacement for the default AST parser. `astroid` offers a richer and more uniform AST structure, and provides access to advanced static analysis utilities, such as type inference and scope resolution. However, these semantic features are exposed only as procedural interfaces and are not embedded directly into the AST itself. As part of our enrichment process, we systematically integrate this semantic information into the ASTs as explicit annotations.

Specifically, we apply the `astypes` library [26] to perform static type inference over the `astroid`-based AST. The inference process combines multiple strategies:

- **Literal inference**: The type of syntactic literals is directly inferred based on their form.
- **Assumption-based inference**: For common Python built-in constructs, such as lists and dictionaries, `astypes` makes practical assumptions based on typical usage patterns.
- **Semantic inference via `astroid`**: `astypes` delegates to `astroid`'s symbolic evaluation when type information cannot be inferred locally.
- **Function return inference**: If the value is determined to be a function call, `astypes` locates the function definition and retrieves its return annotation, if present.
- **`typeshed` fallback**: If the resolved function lacks an explicit annotation, `astypes` queries `typeshed` to retrieve known return types of a standard or built-in functions.

Although limited, this process enables consistent and lightweight type annotation across the entire AST. In addition to type inference, we enrich the AST with reference links based on lexical scope resolution. Specifically, for every `astroid.Name` node, which includes variable, function, or other identifier references, we apply `astroid`'s `lookup` function to resolve all possible symbol bindings visible at that point in the source code. This returns a list of candidate AST nodes corresponding to the entities that the identifier might refer to.

Based on our observation, we annotate each `Name` node with a reference virtual edge (marked as `ref` in the visualization) pointing to the closest matching declaration node among the candidates. This allows us to capture not only variable references, but also function calls and symbolic references to global or nonlocal bindings. In contrast to C++, where Clang's AST provides direct and unambiguous references between usage and declaration nodes via unique `id`-based linkage, Python requires us to reconstruct such connections through static resolution. Our method offers a best-effort approximation of these bindings and embeds them into the AST, enabling richer downstream reasoning just like we have for C++, while accommodating Python's dynamic name resolution semantics.

Similar to our approach for C++, we introduce sequential virtual edges between sibling statements in Python to capture execution order. Together with reference edges, this structure enabled the detection of reference legality by checking whether an identifier is used only after its definition. These lightweight yet expressive augmentations bring Python's dynamic typed AST representation closer to the semantic richness needed for structured code generation.

3.5 Summary

In this project, we design a multi-stage pipeline for converting raw source code into enriched AST representations suitable for structure-aware generative model training. Starting from dataset preprocessing, we extract ASTs for both C++ and Python source code using language-specific tools. We then enrich these ASTs by integrating semantic features that are not directly present in the original syntax trees. Across both languages, we implement a common set of augmentations: type annotations, reference edges linking identifiers to their declarations, and sequential virtual edges encoding statement execution order.

While the enrichment goals are consistent, the implementation differs due to fundamental differences in language design and toolchain capabilities. C++ benefits from a strict and complete type system, and Clang provides precise and fully resolved type information for nearly all expressions at compile time. In contrast, Python lacks static typing by default, and we rely on static program analysis and heuristic assumptions. Despite covering many common cases, this inference process cannot match the completeness or reliability of C++'s type annotations.

Similarly, for identifier resolution, Clang encodes exact symbol bindings directly into the AST through memory-resident identifiers, whereas Python requires scope analysis and approximation. Though this method introduces some ambiguity, it allows for constructing reference edges that are structurally comparable to those in the C++ pipeline.

Despite these differences, our enrichment strategies achieve functionally similar feature sets for both languages. By adapting to the strengths and limitations of each language ecosystem, we produce semantically enriches ASTs with consistent structural features, thus enabling unified training and evaluation across a heterogeneous, multi-language dataset.

4 Implementation and Preliminary Results

This section describes the implementation of our AST enrichment pipeline and presents preliminary results demonstrating its coverage and effectiveness. We discuss the platforms and tooling used to build the prototype, and the range of language features supported in our prototype. Visualizations are also provided to highlight the difference in the AST structures before and after enrichment introduced by our pipeline.

4.1 Pipeline Prototyping

We implemented our AST enrichment pipeline using different toolchains and platforms tailored to each programming languages.

For C++, we invoke the Clang frontend via command-line interface to generate the raw AST in JSON format using the `-Xclang -ast-dump=json` flags. The resulting output is then processed on the .NET platform, where we parse the JSON and reconstruct the AST structure in memory. Each AST node type is mapped to a corresponding .NET object type, and its internal structured is resolved with the guidance of Clang AST's documentation. We additionally recover reference relationships between nodes and prune irrelevant metadata, *e.g.*, source mapping and comment. The final output retains only the topological structure of the AST and the enriched semantic features we introduce.

For Python, all processing is conducted natively within the Python runtime. We use the `astroid` library to parse the source code into an extensible AST structure, and integrate enrichment steps via the `astypes` library and scope tracing as described in Section 3.4.2. The enriched AST is then exported into JSON format using a custom serialization routine, preserving node types, hierarchy, and added semantic annotations.

For both languages, we implement batch processing scripts to automate AST extraction and enrichment across large-scale datasets. In addition, we used the Graphviz toolchain [27] to visualize ASTs before and after enrichment, facilitating both debugging and qualitative comparison. All components of our pipeline, including parsers, enrichers, batch tools and visualization utilities, have been open-sourced on GitHub to support reproducibility and further development [28].

4.2 Language Feature Coverage and Testing

For C++, we compiled a comprehensive list of all AST node types currently supported by our enrichment pipeline. These includes fundamental syntactic constructs such as declarations, control flow statements and expressions. Each supported node type is documented in Table 2, along with a brief description of its function and a link to the corresponding page in the official Clang AST documentation. This table serves both as a reference and as a record of the pipeline's current syntactic coverage.

As noted in Section 3.4.1, we excluded AST nodes related to declarations of classes and lambda expressions due to the complexity of their internal structure and the scope limitations of this prototype. These constructs, while important in general-purpose C++ development, are not frequently used in our dataset and fall outside the focus of our current implementation.

Table 2: Supported C++ AST node types in the enrichment pipeline

Node Type	**Description**	**Ref.**
`ArraySubscriptExpr`	Array Subscripting.	Link
`BinaryOperator`	A builtin binary operation expression such as `x + y` or `x <= y`.	Link
`BindingDecl`	A binding in a decomposition declaration.	Link
`BreakStmt`	This represents a `break`.	Link
`CallExpr`	Represents a function call.	Link
`CaseStmt`	Represent a `case` statement.	Link
`CharacterLiteral`	-	Link
`CompoundAssignOperator`	Keeps track of the type the operation is performed in.	Link
`CompoundStmt`	Represents a group of statements like { `stmt stmt` }.	Link
`ConditionalOperator`	The `?:` ternary operator.	Link
`ConstantExpr`	An expression in a constant context, possibly evaluated.	Link
`ContinueStmt`	This represents a `continue`.	Link
`CStyleCastExpr`	A C-style cast expression.	Link
`CXXBindTemporaryExpr`	Binding an expression to a temporary.	Link
`CXXBoolLiteralExpr`	A boolean literal.	Link
`CXXCatchStmt`	A C++ catch block.	Link
`CXXConstructExpr`	A call to a C++ constructor.	Link
`CXXDefaultArgExpr`	A default argument.	Link
`CXXDefaultInitExpr`	Use of a default initializer.	Link
`CXXDependentScopeMemberExpr`	A member access where the referenced member is unresolved.	Link
`CXXDestructorDecl`	A C++ class destructor.	Link
`CXXForRangeStmt`	C++ ranged `for` loop.	Link
`CXXFunctionalCastExpr`	A cast using functional notation.	Link
`CXXMemberCallExpr`	A call to a member function.	Link
`CXXMethodDecl`	A static or instance method declaration.	Link
`CXXNewExpr`	Represents a `new`-expression.	Link
`CXXNullPtrLiteralExpr`	The null pointer literal.	Link
`CXXOperatorCallExpr`	A call to an overloaded operator.	Link
`CXXStaticCastExpr`	A static_cast expression.	Link
`CXXStdInitializerListExpr`	Construction of `std::initializer_list<T>` from array.	Link
`CXXTemporaryObjectExpr`	C++ cast expression that builds a temporary.	Link
`CXXTryStmt`	A C++ `try` block.	Link
`DeclRefExpr`	A reference to a declared entity.	Link
`DeclStmt`	A declaration as a statement.	Link
`DecompositionDecl`	A decomposition declaration.	Link
`DefaultStmt`	-	Link
`DoStmt`	A `do/while` loop statement.	Link
`EnumConstantDecl`	An `enum` constant definition.	Link
`ExprWithCleanups`	Expression introducing cleanups.	Link
`FloatingLiteral`	-	Link
`ForStmt`	A `for` loop statement.	Link
`FunctionDecl`	A function declaration or definition.	Link
`GNUNullExpr`	GNU __null extension.	Link
`IfStmt`	An `if/then/else` statement.	Link
`ImplicitCastExpr`	Represents an implicit type conversion.	Link
`ImplicitValueInitExpr`	Implicit value initialization.	Link
`InitListExpr`	C/C++ initializer list.	Link
`IntegerLiteral`	-	Link
`MaterializeTemporaryExpr`	Temporary written into memory.	Link

Node Type	Description	Ref.
MemberExpr	Structure and union members.	Link
ParenExpr	A parenthesized expression.	Link
ParmVarDecl	A function parameter.	Link
ReturnStmt	A `return` statement.	Link
StringLiteral	A `string` literal.	Link
SwitchStmt	A `switch` statement.	Link
UnaryExprOrTypeTraitExpr	Expression involving type traits or `sizeof`.	Link
UnaryOperator	A unary expression/operator.	Link
VarDecl	A variable declaration or definition.	Link
WhileStmt	A `while` loop statement.	Link

For Python, our enrichment pipeline supports a broad range of AST node types commonly found in script-based and algorithmic programming. These include module and function definitions, control flow structures, expression, assignments and import statements. Special nodes like `Comprehensions`, `DictComp`'s, `ListComp`'s and `SetComp`'s are also supported. Table 3 lists all supported node types, with the same columns as we have for the C++ AST node type coverage.

Thanks to Python's comparatively simple and uniform syntax, we were able to fully implement enrichment support for both `class` and lambda expression nodes. As a result, our Python-target pipeline implementation achieves near-complete coverage over the node types commonly encountered in the dataset.

Table 3: Supported Python AST node types in the enrichment pipeline

Node Type	Description	Ref.
Assert	Class representing an `ast.Assert` node.	Link
Assign	Class representing an `ast.Assign` node.	Link
AssignAttr	Variation of `ast.Assign` representing assignment to an attribute.	Link
AssignName	Variation of `ast.Assign` representing assignment to a name.	Link
Attribute	Class representing an `ast.Attribute` node.	Link
AugAssign	Class representing an `ast.AugAssign` node.	Link
BinOp	Class representing an `ast.BinOp` node.	Link
BoolOp	Class representing an `ast.BoolOp` node.	Link
Break	Class representing an `ast.Break` node.	Link
Call	Class representing an `ast.Call` node.	Link
ClassDef	Class representing an `ast.ClassDef` node.	Link
Compare	Class representing an `ast.Compare` node.	Link
Comprehension	Class representing an `ast.comprehension` node.	Link
Const	Class representing any constant including num, str, bool, None, bytes.	Link
Continue	Class representing an `ast.Continue` node.	Link
Delete	Class representing an `ast.Delete` node.	Link
Dict	Class representing an `ast.Dict` node.	Link
DictComp	Class representing an `ast.DictComp` node.	Link
ExceptHandler	Class representing an `ast.ExceptHandler` node.	Link
Expr	Class representing an `ast.Expr` node.	Link
For	Class representing an `ast.For` node.	Link
FormattedValue	Class representing an `ast.FormattedValue` node.	Link
FunctionDef	Class representing an `ast.FunctionDef`.	Link
GeneratorExp	Class representing an `ast.GeneratorExp` node.	Link
Global	Class representing an `ast.Global` node.	Link
If	Class representing an `ast.If` node.	Link
IfExp	Class representing an `ast.IfExp` node.	Link
Import	Class representing an `ast.Import` node.	Link
ImportFrom	Class representing an `ast.ImportFrom` node.	Link
JoinedStr	Represents a list of string expressions to be joined.	Link
Lambda	Class representing an `ast.Lambda` node.	Link
List	Class representing an `ast.List` node.	Link
ListComp	Class representing an `ast.ListComp` node.	Link

Node Type	Description	Ref.
`Module`	Class representing an `ast.Module` node.	Link
`Name`	Class representing an `ast.Name` node.	Link
`Nonlocal`	Class representing an `ast.Nonlocal` node.	Link
`Return`	Class representing an `ast.Return` node.	Link
`Set`	Class representing an `ast.Set` node.	Link
`SetComp`	Class representing an `ast.SetComp` node.	Link
`Slice`	Class representing an `ast.Slice` node.	Link
`Starred`	Class representing an `ast.Starred` node.	Link
`Subscript`	Class representing an `ast.Subscript` node.	Link
`Try`	Class representing a `ast.Try` node.	Link
`Tuple`	Class representing an `ast.Tuple` node.	Link
`UnaryOp`	Class representing an `ast.UnaryOp` node.	Link
`While`	Class representing an `ast.While` node.	Link

To quantitatively evaluate the coverage, effectiveness and stability of our enrichment pipeline, we implemented a testing framework that integrates directly with the HuggingFace dataset APIs. This framework allows us to automatically fetch code samples from the original `nvidia/OpenCodeReasoning-2` dataset and apply our full extraction and enrichment pipeline in a reproducible manner.

We conducted separate tests on 1,000 Python and 1,000 C++ code samples from the dataset. Each sample was processed through its respective AST pipeline, and the results were visualized using Graphviz to verify structural correctness and annotation consistency. In the appendix of this report, figures that illustrate representative code samples and their corresponding ASTs before and after enrichment are included. Figures 5 and 6 show the original source code of a C++ sample and a Python sample used for testing. Figures 7 and 9 display the structure of the original plain ASTs generated by Clang AST and `ast` module of the Python language services. In contrast, figures 8 and 10 show the enriched ASTs, where type annotations, reference virtual edges and sequential execution order are incorporated. These visualizations serve as concrete examples of the structural and semantic enhancements designed in Section 3.4 and implemented in Section 4.1.

Out of the 1,000 C++ samples tested, 843 were successfully processed by our pipeline, with their ASR structures fully recognized and enriched. Among the remaining 157 samples, 96 involved unimplemented lambda expression nodes, which were intentionally excluded from our prototype, as noted in Section 3.4.1. 49 sampled failed compilation, which may stem from syntactic or semantic issues in the original code, or the use of language features unsupported by Clang—despite our preprocessing step replacing `bits/stdc++.h` with a list of standard headers to address the majority of such cases. The final 12 samples involved other C++ constructs not yet covered by our pipeline.

In contrast, our Python pipeline achieved significantly higher coverage: 999 out 1,000 samples were parsed, enriched and serialized successfully. The single failure case resulted from a malformed code snippet that could not be parsed by the `astroid` parser.

Specifically, considering the limited ability of the `astypes` library in inferring types for the AST nodes, in the coverage test above, we also gathered statistics for type inference coverage. We implemented type inference for a subset of AST nodes that involves expressions, and for each node type, we counted the occurrences and successful inferences. The results are organized in Table 4.

Table 4: Statistics of type inference in Python AST enrichment

Node Type	# Nodes		#Successful Inferences		Success Rate
	Total	Per Program	Total	Per Program	
`AssignName`	20621	20.62	6984	6.98	33.87%
`Attribute`	6122	6.12	780	0.78	12.74%
`BinOp`	9285	9.29	1996	2.00	21.50%
`BoolOp`	700	0.70	651	0.65	**93.00%**
`Call`	17000	17.00	9370	9.37	55.12%
`Compare`	4915	4.92	4915	4.92	**100.00%**
`Const`	16946	16.95	16946	16.95	**100.00%**
`Dict`	163	0.16	163	0.16	**100.00%**
`DictComp`	15	0.01	15	0.01	**100.00%**
`FormattedValue`	135	0.14	126	0.13	**93.33%**

Node Type	# Nodes		#Successful Inferences		Success Rate
	Total	Per Program	Total	Per Program	
`GeneratorExp`	111	0.11	111	0.11	**100.00%**
`IfExp`	321	0.32	259	0.26	**80.69%**
`JoinedStr`	86	0.09	86	0.09	**100.00%**
`List`	1751	1.75	1751	1.75	**100.00%**
`ListComp`	490	0.49	490	0.49	**100.00%**
`Name`	55346	55.35	14876	14.88	26.88%
`Set`	23	0.02	23	0.02	**100.00%**
`SetComp`	2	0.00	2	0.00	**100.00%**
`Subscript`	7975	7.97	188	0.19	2.36%
`Tuple`	2846	2.85	2846	2.85	**100.00%**
`UnaryOp`	1408	1.41	1316	1.32	**93.47%**
All	146261	146.26	63894	63.89	43.68%

Please note that all AST nodes whose type is `None` are excluded in this statistical result due to the handling mechanism of `asytpes` which returns `None` when it encounters a node whose type cannot be inferred. Combined with this knowledge, data from Table 4 shows that the type inference method we used has excellent outcomes on constant nodes, comparison expressions, system built-in structural types, *e.g.*, `dict` and `set`, and their comparison, unary/binary operation expressions, formatted strings, and conditional expressions (trinomial operators), but has very limited capability on other nodes. At the overall level, (at least) 43.68% of the nodes can be covered by this sort of enrichment.

5 Conclusion and Future Work

The results discussed in Section 4.2 demonstrate the robustness and practical effectiveness of our enrichment pipeline across both languages. The C++ implementation achieves high structural coverage despite the inherent complexity of the language, while the Python pipeline benefits from the language's syntactic simplicity and the flexibility of the toolchain. Together, they validate the feasibility of building multi-language AST enrichment frameworks that adapt to language-specific constraints while producing semantically consistent output representations.

These results confirm the feasibility of AST enrichment as a practical method for constructing semantically informed program representations. By generating structurally consistent enriched ASTs across languages, our pipeline enables the creation of a unified representation space that supports cross-language generalization and provides a suitable foundation for training structure-aware diffusion models.

During the development of this project, we also surveyed a range of existing analysis frameworks and static program analyzers. Parsers and their related toolchains such as ANTLR, JavaCC and Tree-Sitter offer extensible AST extraction capabilities across multiple languages, providing potential alternatives for broader language support in our future work [29]. In addition, large-scale datasets such as Project CodeNet [30], originally created for transformer-based code generation, offer valuable insights into data preparation and representation design. Building on our current pipeline, we plan to explore these tools and resources to find new opportunities of incorporating proper techniques to construct enriched AST datasets at scale, enabling the training of our diffusion models for code generation.

A Test Results

```cpp
#include <iostream>
using namespace std;

int main() {
    long long K;
    while (cin >> K) {
        long long sum = 0;
        for (long long i = 2; i * i ≤ K; ++i) {
            if (K % i == 0) {
                int exponent = 0;
                while (K % i == 0) {
                    exponent++;
                    K /= i;
                }
                sum += exponent * i;
            }
        }
        if (K > 1) {
            sum += K;
        }
        cout << sum << '\n';
    }
    return 0;
}
```

Figure 5: C++ Code Sample `1cc047`

```python
n = int(input())
shop_masks = []
for _ in range(n):
    f = list(map(int, input().split()))
    mask = 0
    for j in range(10):
        if f[j]:
            mask |= 1 << j
    shop_masks.append(mask)
p = []
for _ in range(n):
    p.append(list(map(int, input().split())))
max_total = -float('inf')
for joisino_mask in range(1, 1 << 10):
    total = 0
    for i in range(n):
        overlap = joisino_mask & shop_masks[i]
        c = bin(overlap).count('1')
        total += p[i][c]
    if total > max_total:
        max_total = total
print(max_total)
```

Figure 6: Python Code Sample `336cb2`

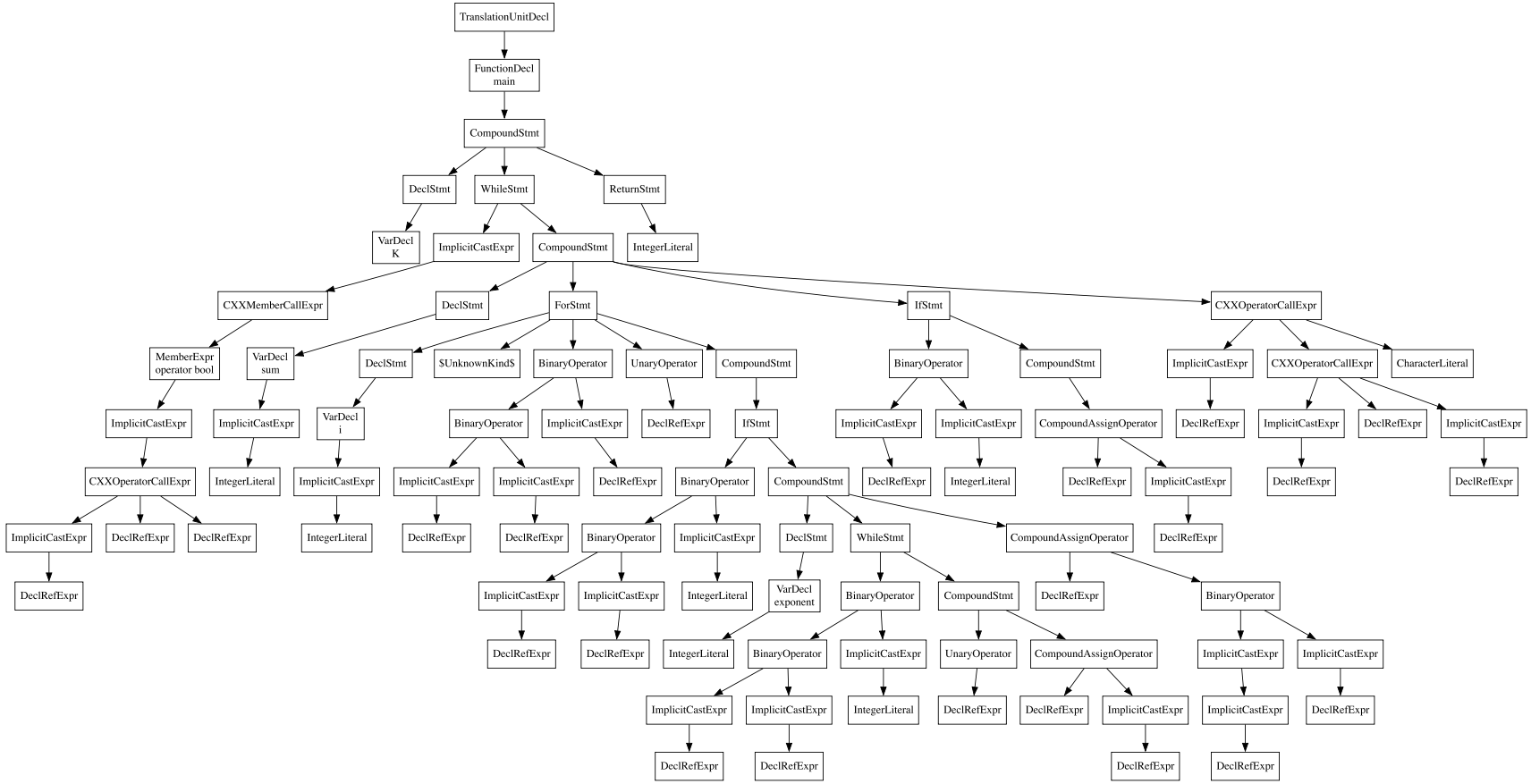


Figure 7: Original AST for C++ Code Sample `1cc047`

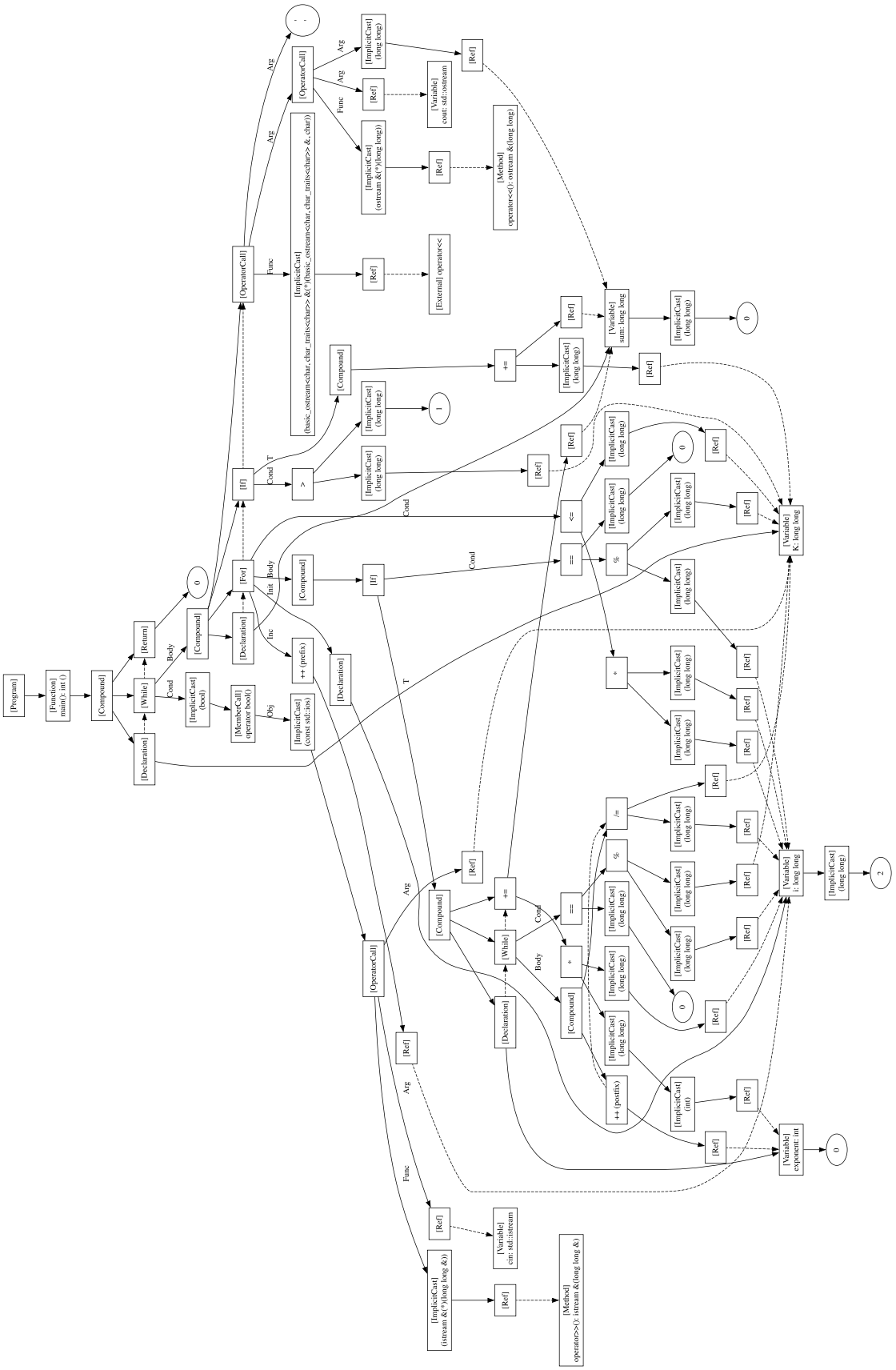


Figure 8: Enriched AST for C++ Code Sample 1cc047

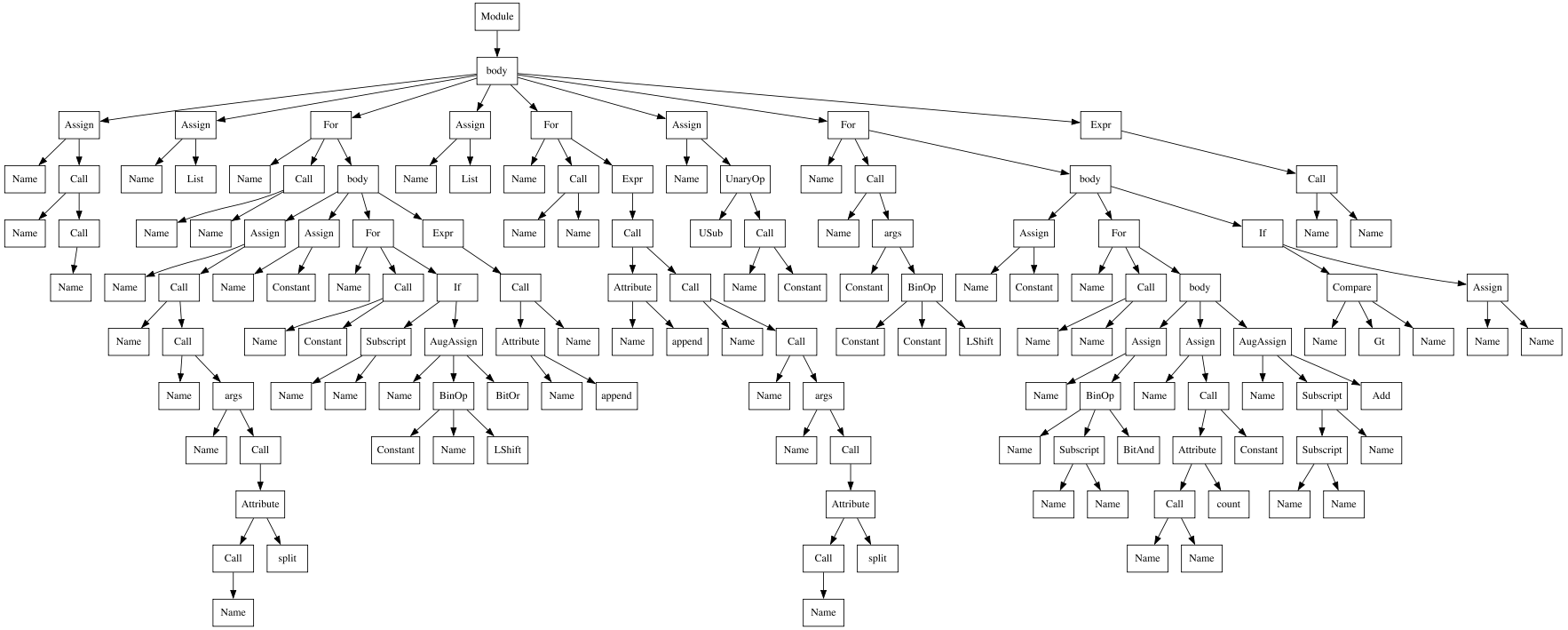


Figure 9: Original AST for Python Code Sample `336cb2`

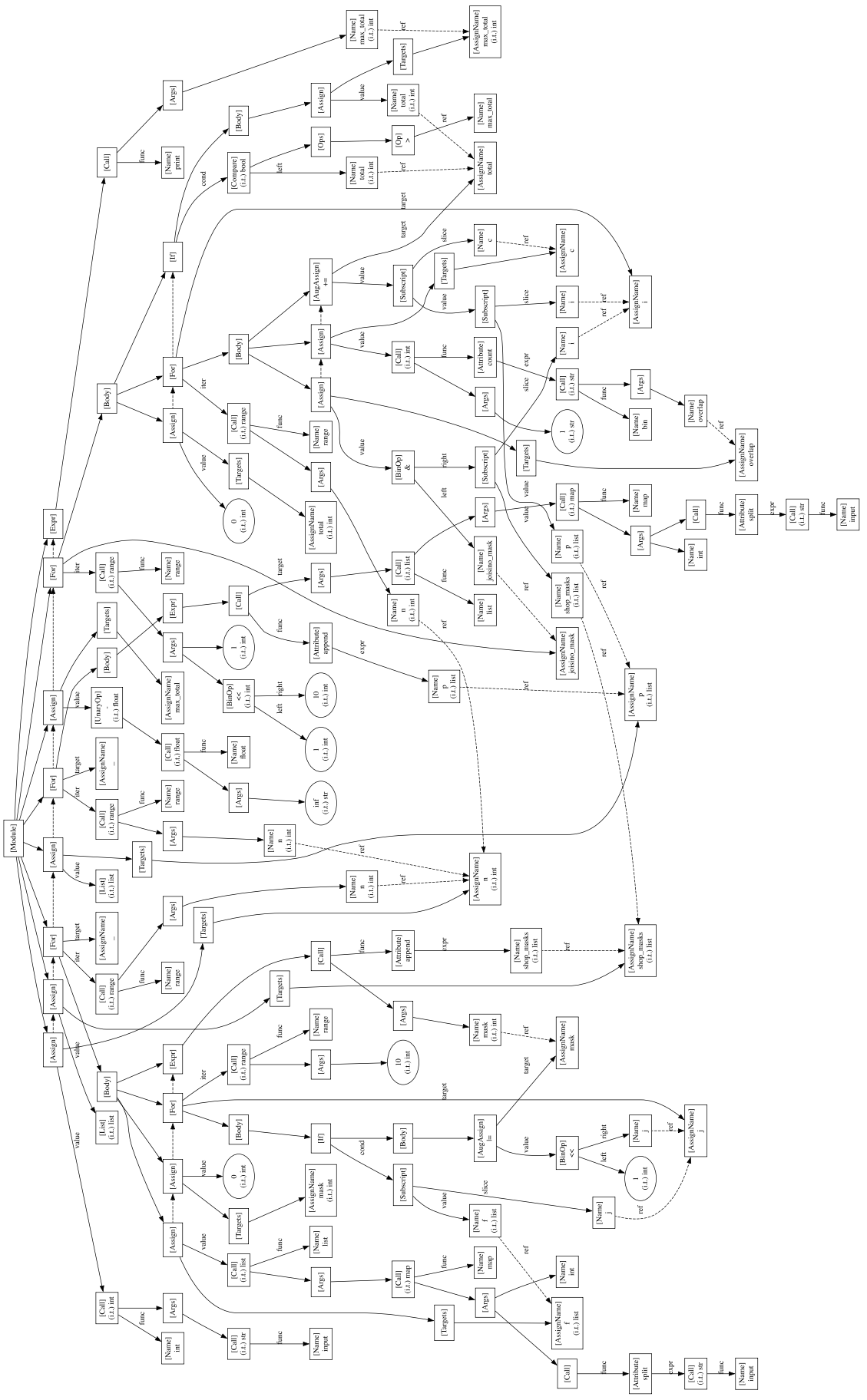


Figure 10: Enriched AST for Python Code Sample 336cb2